Computação de Alto Desempenho (High Performance Computing)

Slides adapted from “An Introduction to Parallel Programming”, Peter Pacheco

Review...

Foster's methodology

1. *Partitioning*

   Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.

2. *Communication*

   Determine what communication needs to be carried out among the tasks identified in the previous step.

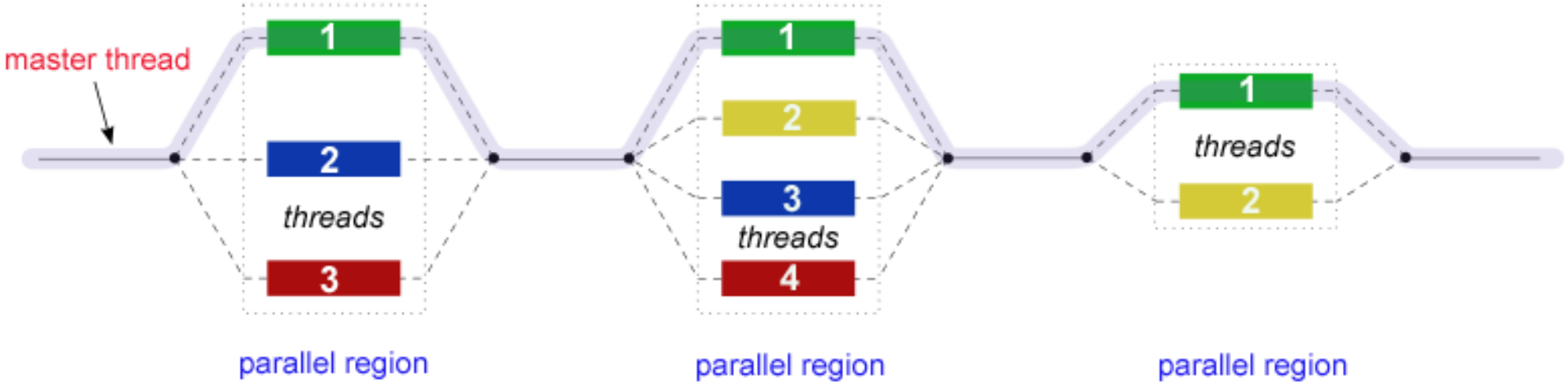
3. *Agglomeration or aggregation*

   Combine tasks and communications identified in the first step into larger tasks.

4. *Mapping*

   Assign the composite tasks identified in the previous step to processes/ threads.

OpenMP Fork-join model



```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
./omp_hello 4
```

compiling

running with 4 threads

possible outcomes

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

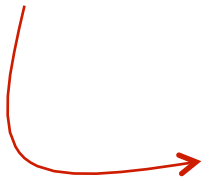
Parallel directive

```
void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```

Reduction clause

A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```



+, *, -, &, |, ^, &&, ||

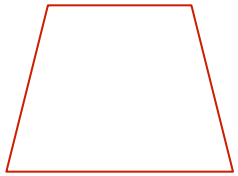
```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

Sections in OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void oneMessage(char c) {
  printf("%c -- Thread:%d Nºthreads:%d\n”, c, omp_get_thread_num(),
omp_get_num_threads());}

int main (int argc, char *argv[])
{
   omp_set_num_threads(2);
  #pragma omp parallel
  {
    #pragma omp sections
     {
        #pragma omp section
        { oneMessage(‘A');}
        #pragma omp section
        { oneMessage(‘B'); }
    }
   }
    return(0);
}
```

Parallel for and variables' scope

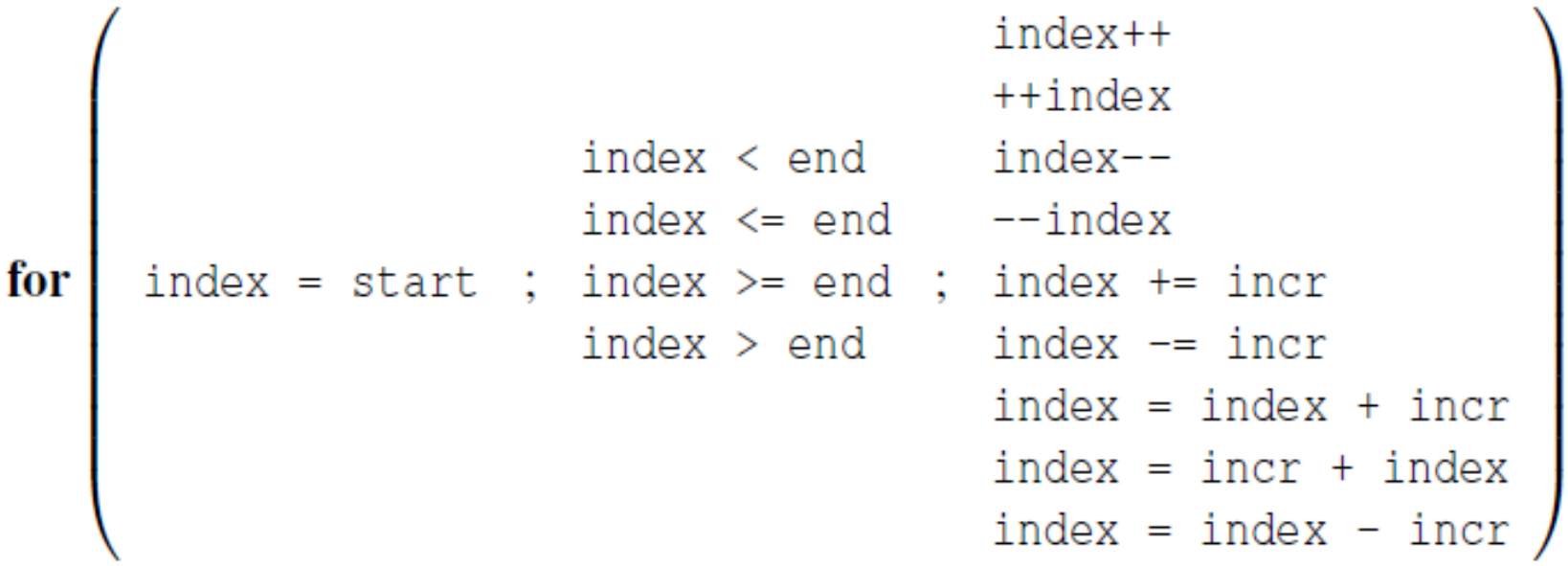
```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```



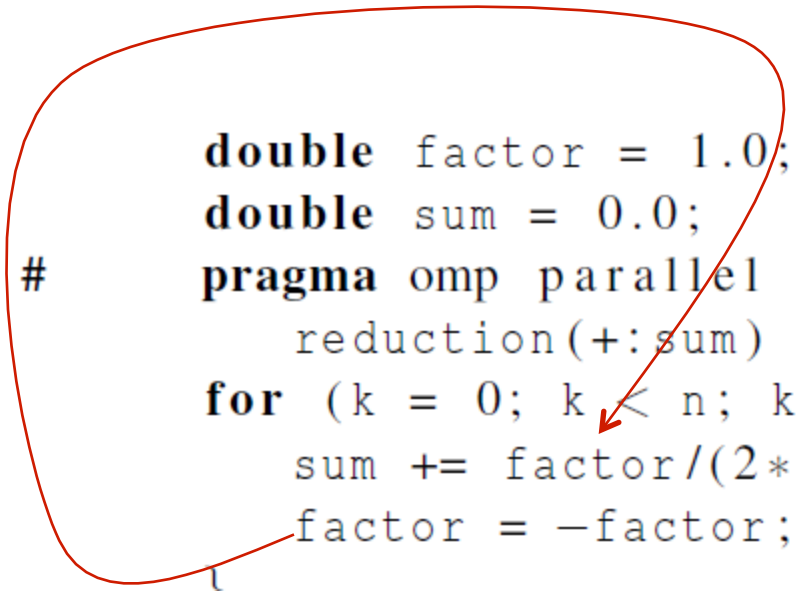
```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```

The varibles' **default scope** is **private**; *i* is private; there are local variables being used to add to the **reduction variable** *approx*

Legal forms for *parallelizable for statements*



Loop dependency




```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum)
        for (k = 0; k < n; k++) {
            sum += factor/(2*k+1);
            factor = -factor;
        }
        pi_approx = 4.0*sum;
```

The default clause

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           default(none) reduction(+:sum) private(k, factor) \
           shared(n)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

MORE ABOUT LOOPS IN OPENMP: SORTING

Serial bubble sort

```
void Bubble_sort(
        int  a[]  /* in/out */,
        int  n    /* in     */) {
    int list_length, i, temp;

    for (list_length = n; list_length >= 2; list_length--)
        for (i = 0; i < list_length-1; i++)
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }

}  /* Bubble_sort */
```

Bubble sort

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

Is this algorithm parallelizable?

Bubble sort

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

There is an inherently sequential ordering of the comparisons.
Ex:
a[i-1] = 9, a[i] = 5, and a[i+1] = 7

If different comparison orders were possible:
a) Compare 9 and 5 and swap them,
compare 9 and 7 and swap them → **5, 7, 9**

b) compare the 5 and 7 first and then
compare the 9 and 5 → **5, 9, 7**

The order in which the "compare-swaps" take place is essential to the correctness of the algorithm.

Bubble sort

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

Loop-carried dependence in the inner loop:
-- the elements compared in iteration i depend on on the outcome of iteration i−1

Ex: list= a= {3,1,2}
i = 1 → *compare 3 and 2*

But if the *i = 0 and the i = 1* iterations are happening *simultaneously* → it may happen: *compare 1 and 2*.

Bubble sort

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

Loop-carried dependence in the outer loop: in any iteration of the outer loop the contents of the current list depends on the previous iterations of the outer loop.
Ex:
a = {3, 4, 1, 2}; the second iteration should act on {3, 1, 2} (4 should be at the last position of a).
But this may not happen if the first two iterations are executing simultaneously (4 may still be in the list).

Odd-even transposition sort

- A variant of *bubble sort* with more opportunities for parallelism
  - key idea: to "decouple" the compare-swaps

Odd-even transposition sort

- A sequence of phases.
- Even phases, compare swaps are executed on:

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots$$

- Odd phases, compare swaps are executed on:

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots.$$

Example

Start: 5, 9, 4, 3

Even phase: compare-swap (5,9) and (4,3)
getting the list 5, 9, 3, 4

Odd phase: compare-swap (9,3)
getting the list 5, 3, 9, 4

Even phase: compare-swap (5,3) and (9,4)
getting the list 3, 5, 4, 9

Odd phase: compare-swap (5,4)
getting the list 3, 4, 5, 9

Theorem

- Suppose A is a list with n keys/values, and A is the input to the odd-even transposition sort algorithm. Then, after n phases A will be sorted.

Serial odd-even transposition sort

```
void Odd_even_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int phase, i, temp;

   for (phase = 0; phase < n; phase++)
      if (phase % 2 == 0) { /* Even phase */
         for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) {
               temp = a[i];
               a[i] = a[i-1];
               a[i-1] = temp;
            }
      } else { /* Odd phase */
         for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) {
               temp = a[i];
               a[i] = a[i+1];
               a[i+1] = temp;
            }
      }
}  /* Odd_even_sort */
```

Serial odd-even transposition sort

```
void Odd_even_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int phase, i, temp;

   for (phase = 0; phase < n; phase++)
      if (phase % 2 == 0) { /* Even phase */
         for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) {
               temp = a[i];
               a[i] = a[i-1];
               a[i-1] = temp;
            }
      } else { /* Odd phase */
         for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) {
               temp = a[i];
               a[i] = a[i+1];
               a[i+1] = temp;
            }
      }
} /* Odd_even_sort */
```

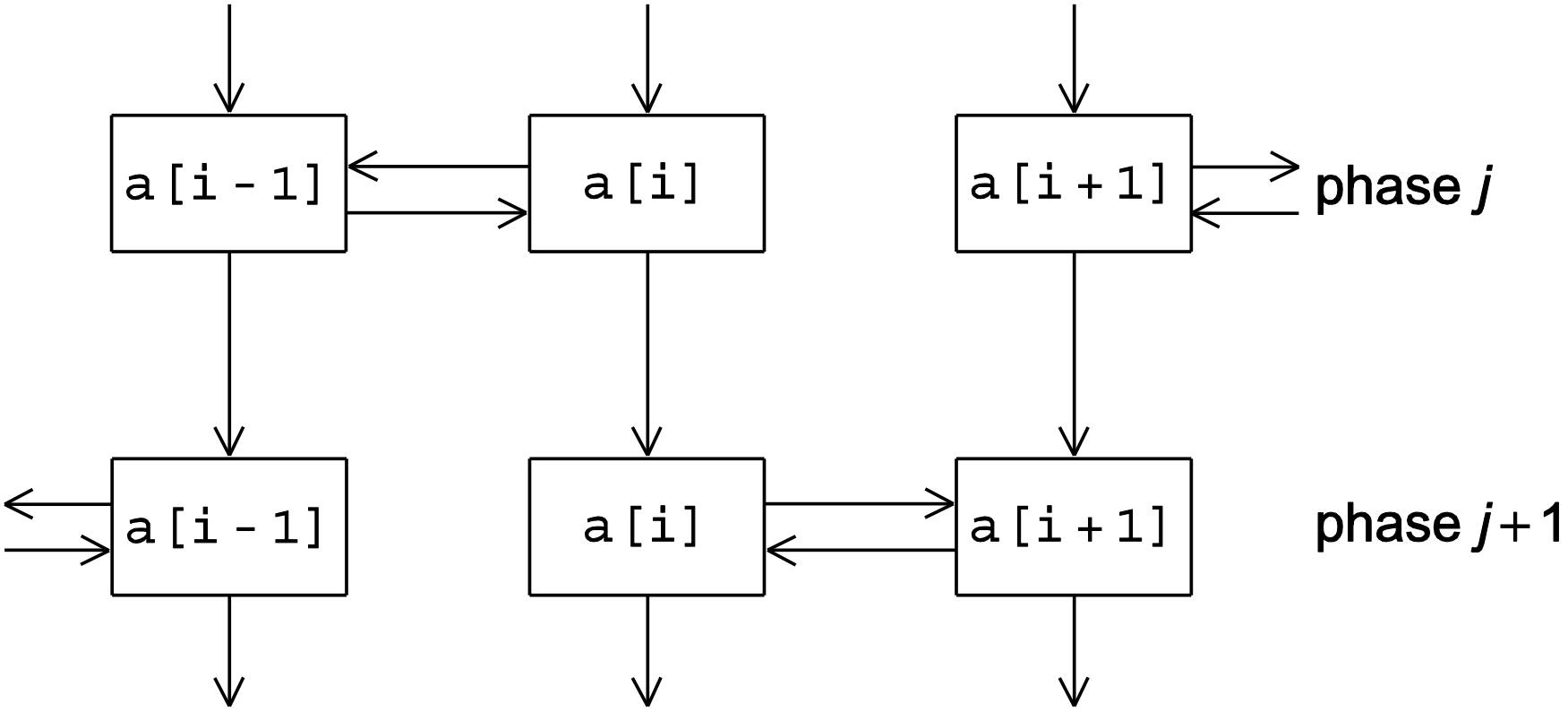
All of the compare-swaps in a single phase can happen simultaneously

Serial odd-even transposition sort

- *Tasks*: Determine the value of a[i] at the end of phase j.
- *Communications/Dependencies*: The task that is determining the value of a[i] needs to communicate with either the task determining the value of a[i-1] or a[i+1].

  Also the value of a[i] at the end of phase j needs to be available for determining the value of a[i] at the end of phase j+1.

Dependencies/communications among tasks in odd-even sort



Tasks determining/defining the value of a[i] are labeled with a[i].

Serial odd-even transposition sort

```
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
    else
        for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

Serial odd-even transposition sort

Phase	Subscript in Array							
	0		1		2		3	
0	9	↔	7		8	↔	6	
	7		9		6		8	
1	7		9	↔	6		8	
	7		6		9		8	
2	7	↔	6		9	↔	8	
	6		7		8		9	
3	6		7	↔	8		9	
	6		7		8		9	

a = {9, 7, 8, 6}

Serial odd-even transposition sort

Phase	Subscript in Array							
	0		1		2		3	
0	9	↔	7		8	↔	6	
	7		9		6		8	
1	7		9	↔	6		8	
	7		6		9		8	
2	7	↔	6		9	↔	8	
	6		7		8		9	
3	6		7	↔	8		9	
	6		7		8		9	

a = {9, 7, 8, 6}

Parallelization?

The outer loop has a loop-carried dependence. E.g. phase 0 and phase 1 cannot be executed simultaneously.

The inner loops do not seem to have any loop-carried dependences

First OpenMP odd-even sort

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

First OpenMP odd-even sort

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Problems?

First OpenMP odd-even sort

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Problem1: All the threads have to finish phase p before any thread starts phase p+1.

Parallel for has ***an implicit barrier*** → main thread waits for all threads in phase p

First OpenMP odd-even sort

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Problem 2: overhead associated with forking and joining the threads -- the OpenMP implementation may fork and join *thread_count* threads on each pass

First OpenMP odd-even sort

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Same number of threads for each inner loop → fork the threads once and reuse the same team of threads for each execution of the inner loops

Second OpenMP odd-even sort

```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Fork a team of thread count threads ***before*** the outer loop with a ***parallel directive***

Second OpenMP odd-even sort

```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Fork a team of thread count threads ***before*** the outer loop with a ***parallel directive***

For directive -- tells OpenMP to parallelize the for loop with the existing team of threads

Second OpenMP odd-even sort

```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Fork a team of thread count threads ***before*** the outer loop with a ***parallel directive***

For directive -- tells OpenMP to parallelize the for loop with the existing team of threads

Odd-even sort with two parallel for directives and two for directives. (Times are in seconds.)

Input list with 20,000 elements.

thread_count	1	2	3	4
Two `parallel` **for** directives	0.770	0.453	0.358	0.305
Two **for** directives	0.732	0.376	0.294	0.239

SCHEDULING LOOPS

We want to parallelize this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

Thread	Iterations
0	$0, n/t, 2n/t, \ldots$
1	$1, n/t+1, 2n/t+1, \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1, n/t+t-1, 2n/t+t-1, \ldots$

Assignment of work using cyclic partitioning.

```
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}  /* f */
```

Our definition of function *f*.

```
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}  /* f */
```

Our definition of function *f*.

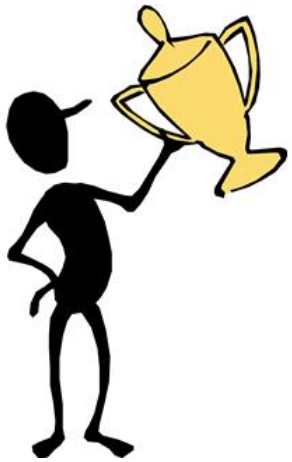
Results

- f(i) calls the sin function *i* times.
- Assume the time to execute f(2i) requires approximately twice as much time as the time to execute f(i).

- n = 10,000
  - one thread
  - run-time = 3.67 seconds.

Results

- n = 10,000
  - two threads
  - default assignment
  - run-time = 2.76 seconds
  - speedup = 1.33
- n = 10,000
  - two threads
  - cyclic assignment
  - run-time = 1.84 seconds
  - speedup = 1.99



The schedule clause

- Default schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- Cyclic schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.

The static schedule type

twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,1)
```

Thread 0 : 0, 3, 6, 9

Thread 1 : 1, 4, 7, 10

Thread 2 : 2, 5, 8, 11

The static schedule type

twelve iterations, 0, 1, . . . , 11, and three threads

`schedule(static,2)`

Thread 0 :	0, 1, 6, 7
Thread 1 :	2, 3, 8, 9
Thread 2 :	4, 5, 10, 11

The static schedule type

twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,4)
```

$$\begin{aligned} \text{Thread } 0: &\quad 0,1,2,3 \\ \text{Thread } 1: &\quad 4,5,6,7 \\ \text{Thread } 2: &\quad 8,9,10,11 \end{aligned}$$

The dynamic schedule type

- The iterations are also broken up into chunks of chunksize consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- This continues until all the iterations are completed.
- The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.

The guided schedule type

- Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
- However, in a guided schedule, as chunks are completed the size of the new chunks decreases.
- If no chunksize is specified, the size of the chunks decreases down to 1.
- If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

The runtime schedule type

- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.
- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.
  - E.g.

    $ export OMP_SCHEDULE="static,1”
  - By using the *schedule(runtime)* clause, different options can be explored with different assignments to OMP_SCHEDULE.

PRODUCERS AND CONSUMERS

Queues

- Can be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket.
- A natural data structure to use in many multithreaded applications.
- For example, suppose we have several “producer” threads and several “consumer” threads.
  - Producer threads might “produce” requests for data (e.g. from a server providing current stock market values).
  - Consumer threads might “consume” the request by finding or generating the requested data (e.g. the stock data).

Message-passing

- Shared memory can be used to implement message-passing
- Each thread could have a shared message queue, and when one thread wants to "send a message" to another thread, it could enqueue the message in the destination thread's queue.
- A thread could receive a message by dequeuing the message at the head of its message queue.

Message-passing

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();
    Try_receive();
}

while (!Done())
    Try_receive();
```

Each thread alternates between sending and receiving messages.

When a thread does not have any message to send, waits for all the other threads to finish (receives the messages from the other threads)

Sending messages -- Send_msg() pseudo-code

```
    mesg = random();
    dest = random() % thread_count;
#   pragma omp critical
    Enqueue(queue, dest, my_rank, mesg);
```

Destination thread (owner of the queue)

Identifier of the sending thread

Only the owner of a queue can dequeue a message, but several threads can enqueue their own messages on this Queue.

Receiving messages – Try_receive() pseudo-code

queue_size = enqueue - dequeue

```
    if (queue_size == 0) return;
    else if (queue_size == 1)
#       pragma omp critical
        Dequeue(queue, &src, &mesg);
    else
        Dequeue(queue, &src, &mesg);
    Print_message(src, mesg);
```

Termination detection

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```

each thread increments this after completing its for loop

Startup (1)

- When the program begins execution, a single thread, the master thread, will get command line arguments and allocate an array of message queues: one for each thread.
- This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.

Startup (2)

- One or more threads may finish allocating their queues before some other threads.
- We need an explicit barrier so that when a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier.
- After all the threads have reached the barrier all the threads in the team can proceed.

```
# pragma omp barrier
```

The atomic directive (1)

- Unlike the critical directive, it can only protect critical sections that consist of a single C assignment statement.

```
# pragma omp atomic
```

- Further, the statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

The atomic directive (2)

- Here <op> can be one of the binary operators

```
+, *, -, /, &, ^, |, <<, or >>
```

- Many processors provide a special load-modify-store instruction.
- A critical section that only does a load-modify-store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

Critical sections

- OpenMP provides the option of adding a name to a critical directive:

```
# pragma omp critical(name)
```

- When we do this, two blocks protected with critical directives with different names can be executed simultaneously.
- However, the names are set during compilation, and we want a different critical section for each thread's queue.

Locks

- A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section.

Locks

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```

Using locks in the message-passing program

```
#  pragma omp critical
   /* q_p = msg_queues[dest] */
   Enqueue(q_p, my_rank, mesg);
```

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

Using locks in the message-passing program

```
# pragma omp critical
/* q_p = msg_queues[my_rank] */
Dequeue(q_p, &src, &mesg);
```

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

Some caveats

1. You should not mix the different types of mutual exclusion for a single critical section.
2. There is no guarantee of fairness in mutual exclusion constructs.
3. It can be dangerous to “nest” mutual exclusion constructs.

Concluding Remarks (1)

- OpenMP is a standard for programming shared-memory systems.
- OpenMP uses both special functions and preprocessor directives called pragmas.
- OpenMP programs start multiple threads rather than multiple processes. E.g. most general

```
# pragma omp parallel
    structured block
```

- Many OpenMP directives can be modified by clauses.

Concluding Remarks (2)

- A major problem in the development of shared memory programs is the possibility of race conditions.
- OpenMP provides several mechanisms for insuring mutual exclusion in critical sections.
  - Critical directives
  - Named critical directives
  - Atomic directives
  - Simple locks

Concluding Remarks (3)

- Critical directives

  # pragma omp critical
  structured block

- Atomic directives

  x <op>= <expression>

- Default behavior of OpenMP — all critical blocks are treated as part of one composite critical section, i.e. only one thread at a time -- this is valid both for unnamed critical directives and atomic directives

```
# pragma omp atomic
    x++;

# pragma omp atomic
    y++;
```

  - This can be highly detrimental to a program's performance

Concluding remarks (4)

- Named critical directives -- threads entering critical sections with different names can execute concurrently.

  # pragma omp critical(name)

- Simple locks

  omp set lock(&lock);

  critical section

  omp unset lock(&lock);

Concluding remarks (5)

- You should not mix the different types of mutual exclusion for a single critical section

```
# pragma omp atomic          # pragma omp critical
  x += f(y);                   x = g(x);
```

The critical directive does not exclude the action executed by the atomic block.

Use critical on both cases or rewrite g(x)

Concluding remarks (6)

- There is no guarantee of fairness in mutual exclusion constructs

```
while(1) {
   . . .
   # pragma omp critical
   x = g(my rank);
   . . .
}
```

One thread may block forever. This does not happen if threads are scheduled in a round-robin fashion, or if

a *for loop* is used.

Concluding remarks (7)

- It can be dangerous to "nest" mutual exclusion constructs

```
# pragma omp critical
y = f(x);
. . .
double f(double x) {
  # pragma omp critical
      z = g(x); /* z is shared */
    . . .
}
```

  - A thread will block here, as all other threads

Concluding remarks (8)

… solve the anterior problem by using named critical sections

```
# pragma omp critical(one)
y = f(x);
. . .
double f(double x) {
   # pragma omp critical(two)
       z = g(x); /* z is global */
       . . .
}
```

However this does not work in some situations → deadlock

Deadlock...

Time	Thread *u*	Thread *v*
0	Enter crit. sect. `one`	Enter crit. sect. `two`
1	Attempt to enter `two`	Attempt to enter `one`
2	Block	Block

Concluding remarks (9)

- By default most systems use a block-partitioning of the iterations in a parallelized for loop.
- OpenMP offers a variety of scheduling options.

  schedule(<type> [,<chunksize>])
- In OpenMP the scope of a variable is the collection of threads to which the variable is accessible.

Concluding remarks (10)

- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- A barrier directive will cause the threads in a team to block until all the threads have reached the directive.

```
# pragma omp barrier
```

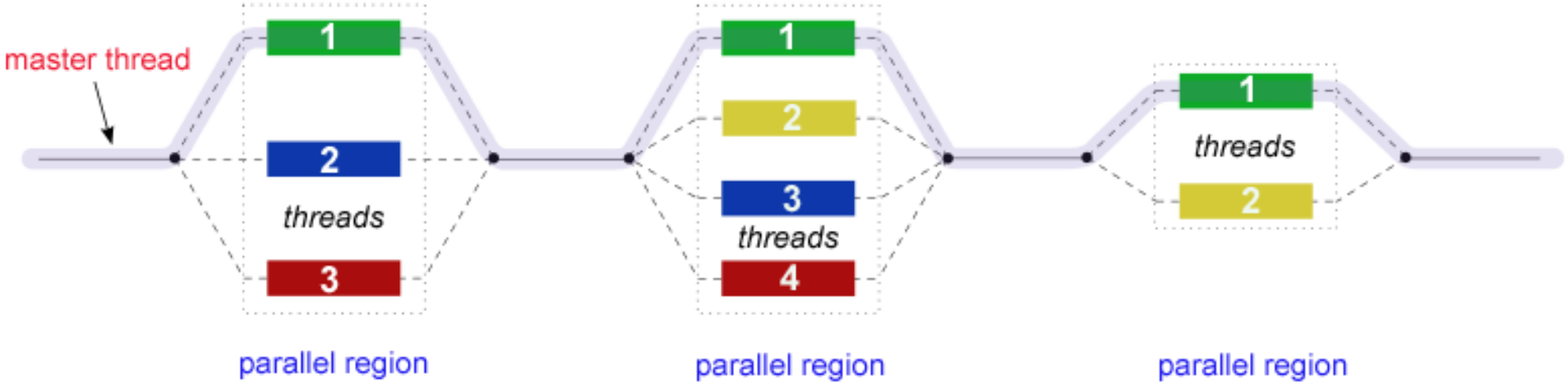
  - The parallel, parallel for, and for directives have implicit barriers at the end of the structured block.

Tutorial OpenMP

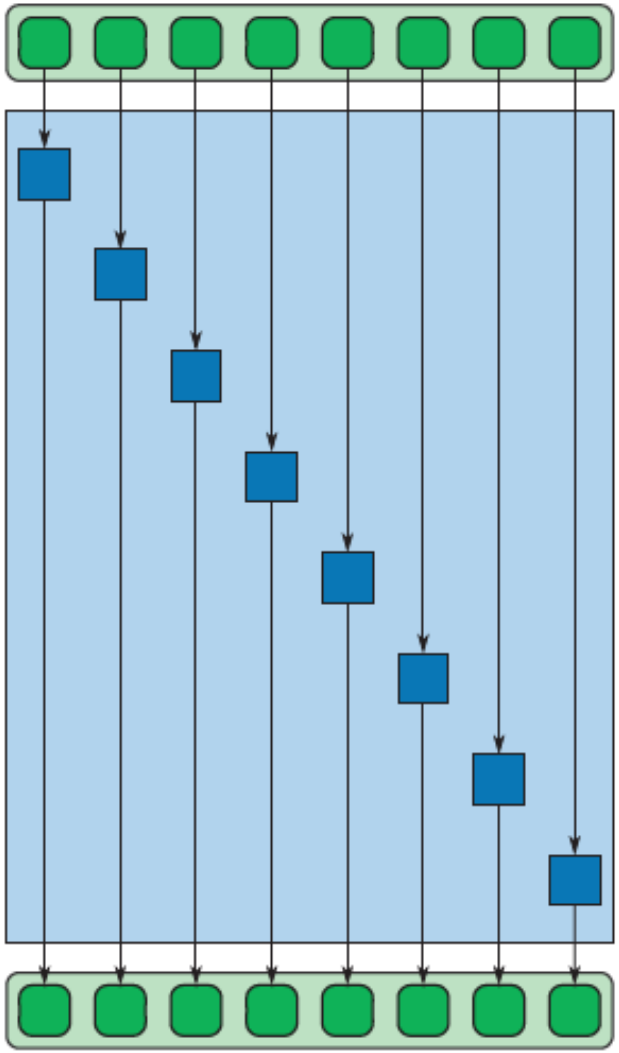
https://computing.llnl.gov/tutorials/openMP/

Common patterns…

Fork-join model



Map- serial execution



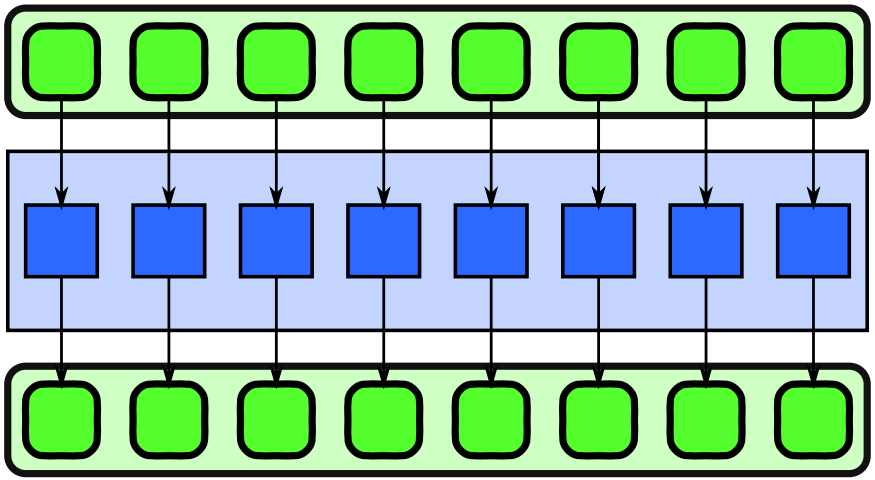
- *Map* replicates a function over every element of an index set
- The index set may be abstract or associated with the elements of an array.

```
A = map(f)(B);
```

```
for(i=0; i<len; i++)
  a[i] = a[i] + 0x20
```

M. McCool, A. Robision, J. Reinders

Map – parallel execution



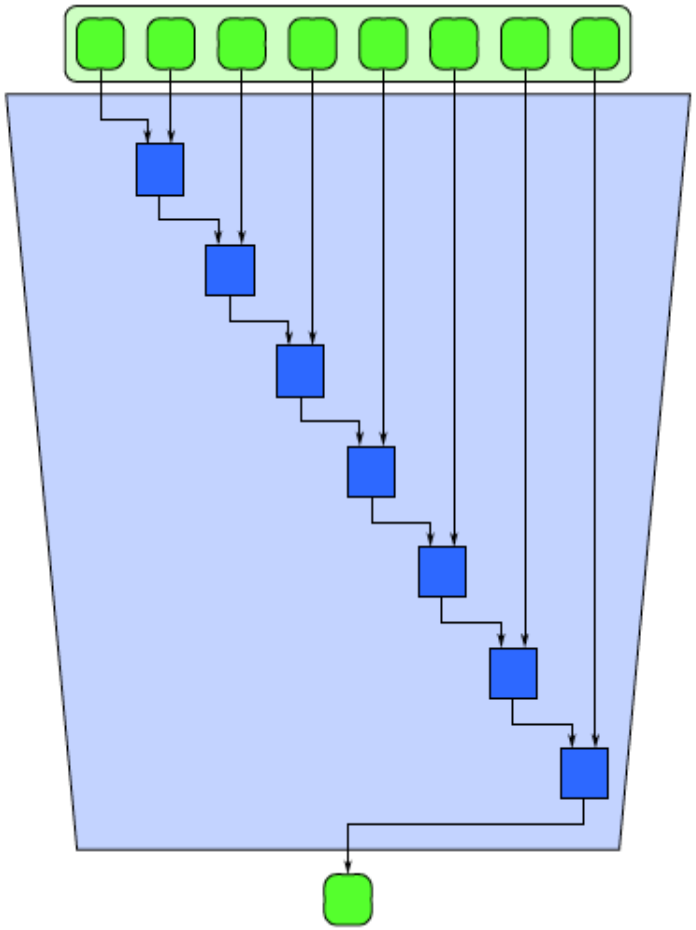
- *Map* replicates a function over every element of an index set
- The index set may be abstract or associated with the elements of an array.

```
A = map(f)(B);
```

- Map replaces *one specific* usage of iteration in serial programs: independent operations.

M. McCool, A. Robision, J. Reinders

Reduction – serial execution



- *Reduce* combines every element in a collection into one element using an associative operator.

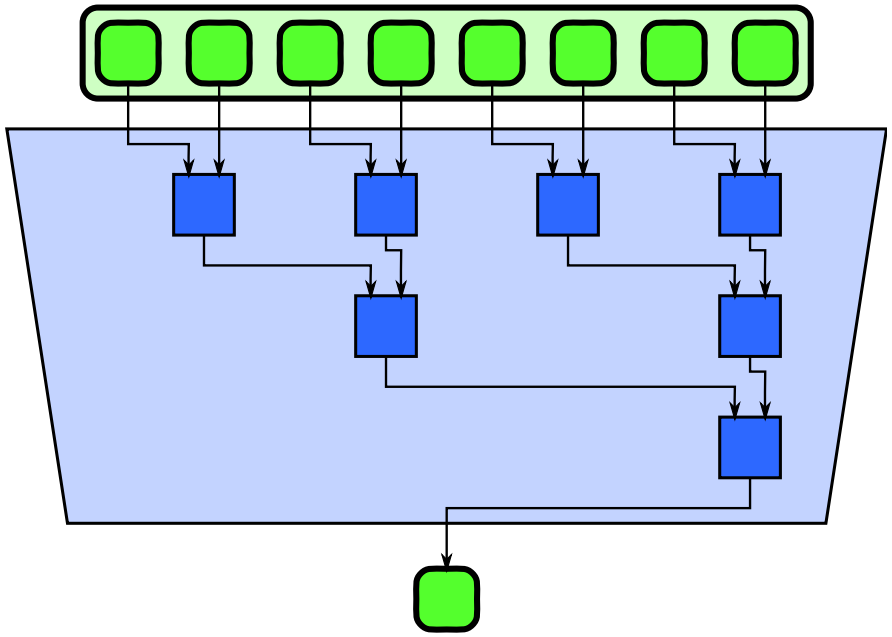
```
b = reduce(f)(B);
```

- For example: *reduce* can be used to find the sum or maximum of an array.

```
for(i=0; i<len; i++)
    total += a[i]
```

M. McCool, A. Robision, J. Reinders

Reduction – parallel execution



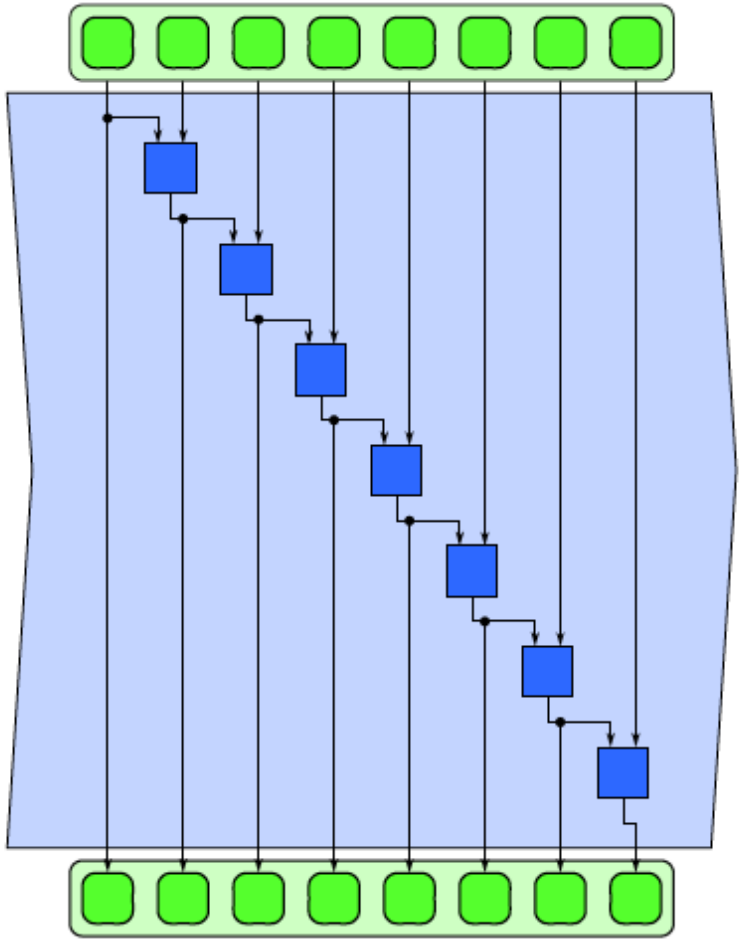
- *Reduce* combines every element in a collection into one element using an associative operator.

```
b = reduce(f)(B);
```

- For example: *reduce* can be used to find the sum or maximum of an array.

M. McCool, A. Robision, J. Reinders

Scan – serial execution



- *Scan* computes all partial reductions of a collection

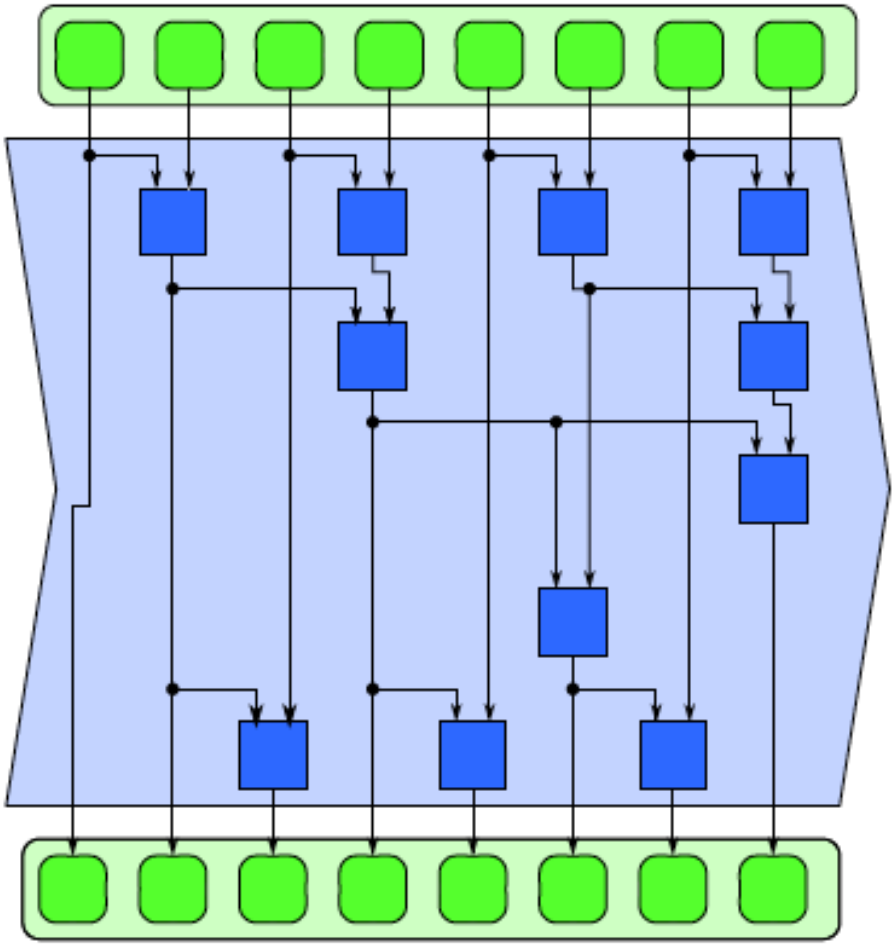
```
A = scan(f)(B);
```

- Operator must be (at least) associative.

```
for(i=0; i<len-1; i++)
    a[i+1] = a[i] + a[i+1]
```

M. McCool, A. Robision, J. Reinders

Scan – parallel execution



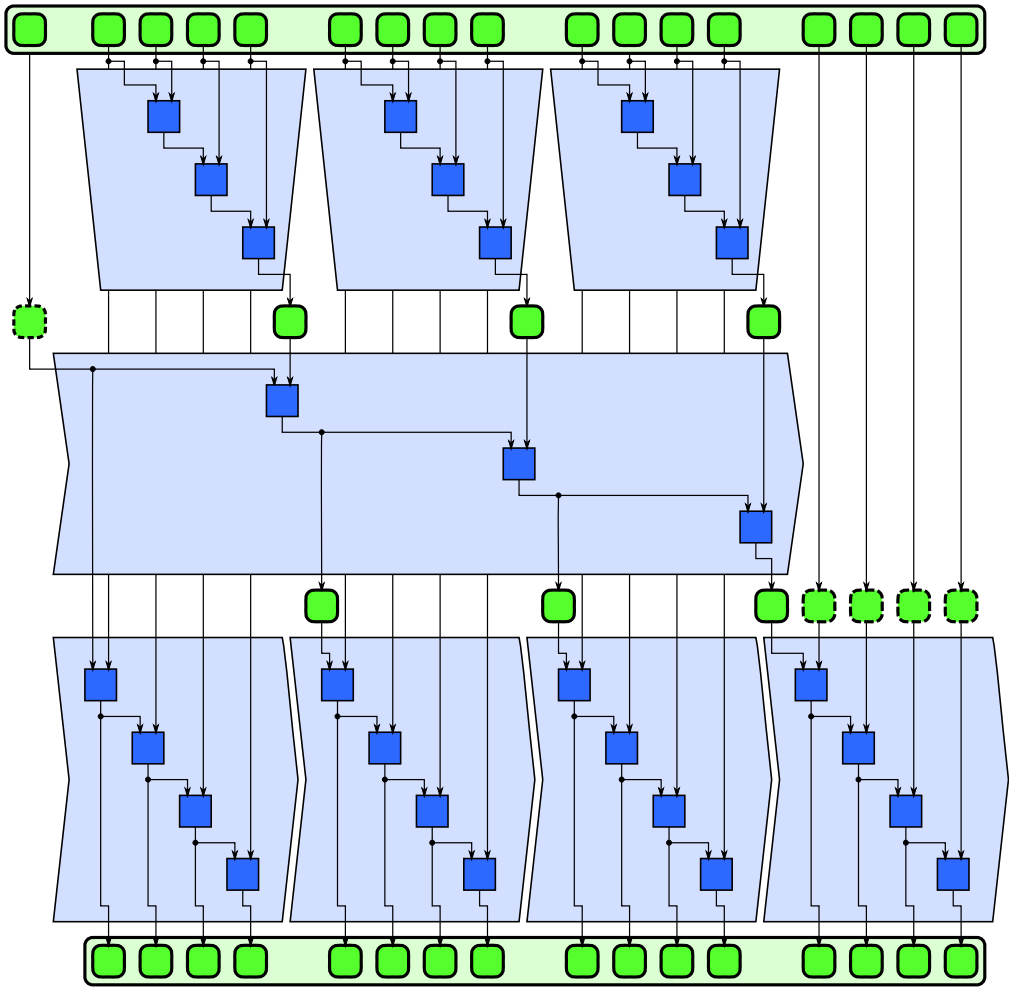
- *Scan* computes all partial reductions of a collection

```
A = scan(f)(B);
```

- Operator must be (at least) associative.

M. McCool, A. Robision, J. Reinders

Scan – parallel execution



- *Scan* computes all partial reductions of a collection

```
A = scan(f)(B);
```

- Operator must be (at least) associative.
- Diagram shows one possible parallel implementation using three-phase strategy